INRIA

UNITÉ DE RECHERCHE
INRIA-SOPHIA ANTIPOLIS

Institut National
de Recherche
en Informatique
et en Automatique

Domaine de Voluceau
Rocquencourt
B.P. 105
78153 Le Chesnay Cedex
France
Tél. (1) 39 63 55 11

Rapports Techniques

N° 115

Programme 1
Programmation, Calcul Symbolique
et Intelligence Artificielle

UN GROUPEMENT DE COMMANDES UNIX POUR LE TRI DES ENREGISTREMENTS MULTILIGNES

Philippe VERDRET

Décembre 1989

RT_0115

UN GROUPEMENT DE COMMANDES *UNIX* POUR LE TRI DES ENREGISTREMENTS MULTILIGNES

A *UNIX* COMMAND SET FOR MULTILINE-RECORD SORTING

Philippe VERDRET

INRIA Sophia Antipolis
2004 route des Lucioles
B.P. 109
06561 Valbonne Cedex
FRANCE

Résumé

Dans ce rapport nous présentons un groupement de commandes UNIX permettant de réaliser simplement le tri d'enregistrements multilignes. La méthode proposée est générale et susceptible d'être appliquée à diverses opérations sur un ensemble d'enregistrements multilignes. Pour illustrer sa mise en œuvre nous l'appliquerons à un problème pour lequel elle est particulièrement bien adaptée, celui de la construction de la bibliographie d'un document à partir des clés bibliographiques qu'il contient.

Abstract

This report presents a UNIX command set that allows simple multiline-record sorting. The proposed method is general : it can be used to perform various operations on a set of multiline records. As an illustration, the method is applied to a problem for which it is well suited : the problem of building the reference section of a document from its bibliographical keys.

Table des matières

"... virtually every *important aspect of programming arises somewhere in the context of sorting or searching ... "*
Knuth, D.E. [Knuth 73]

1 Introduction

Les possibilités des utilitaires UNIX pour réaliser des opérations de traitement de texte et de gestion de base de données ont été maintes fois illustrées [Aho 88], [Aho 86], [Bentley 85]. Ce rapport décrit un groupement d'utilitaires permettant de réaliser simplement le tri des enregistrements multilignes. Dans le monde UNIX ce problème n'a pas de solution immédiate, la commande de tri du système — **sort** — ne réalisant qu'un tri sur enregistrement monoligne. Une méthode pour résoudre ce problème consiste à joindre les lignes d'un enregistrement le temps d'appliquer la commande **sort** (voir [Aho 88]). La méthode proposée ici repose sur un principe différent. Elle consiste à ne trier que les clés et à générer, à partir de ces clés, un script de commandes dont la fonction est de transmettre le tri aux enregistrements associés aux clés.

Dans ce rapport nous appliquerons la méthode à un problème pour lequel elle est particulièrement bien adaptée : la construction de la bibliographie d'un document. Après extraction du document de l'ensemble des clés renvoyant à des références bibliographiques, la méthode permet de trouver les références contenues dans une base de données et de les placer en ordre alphabétique dans un fichier bibliographique devant accompagner le document.

Nous verrons que la méthode est générale puisque susceptible d'être mise en œuvre pour résoudre d'autres problèmes que celui traité ici. Elle est implantée à l'aide d'un groupement de commandes — ou *script* — qui utilise les interprètes de commandes **ksh**, **sed** et **awk**. Le premier est une version du shell de UNIX, les deux derniers sont des utilitaires qui permettent de réaliser des opérations d'édition non interactives sur des fichiers de texte. Le script présenté illustre particulièrement bien les possibilités de programmation offertes par ces deux utilitaires : il est simple, concis et rapide à mettre en œuvre.

Nous commencerons par décrire précisément le problème posé et les solutions envisageables. Puis nous décrirons les outils UNIX utilisés. Le lecteur familier avec ces outils pourra passer directement à la dernière partie du rapport consacrée au groupement proposé. Pour présenter le groupement, nous partirons d'une version minimale. Nous montrerons ensuite comment il est possible de l'optimiser et de l'adapter à des situations spécifiques de mise en œuvre.

2 Le problème et ses solutions

La méthode proposée n'est absolument pas liée au seul domaine LaTeX que nous utilisons pour illustrer sa mise en œuvre. En LaTeX, les enregistrements ont la forme présentée à la figure 1. Ils se composent d'une ou plusieurs lignes de textes. Nous supposerons qu'ils

doivent être séparés par au minimum une ligne vide, ce qui est un prérequis pour la mise en œuvre de notre méthode. Les clés que nous utiliserons pour le tri figurent entre {kn: et }.

```
\bibitem[Aho 88]{kn:Aho_al88} {\sc Aho, A.V., Kernighan, B.W. \&
Weinberger, P.J.} :
{\em The awk Programming Languages}.
Addison-Wesley Publishing Compagny, 1988.
        .
        .
        .
\bibitem[Bentley 85]{kn:Ben85} {\sc Bentley, J.} :
Programming Pearls.
{\it Communications of the ACM},
{\bf 28}(6), Jun. 1985.
        .
        .
        .
```

Figure 1: références contenues dans une bibliographie au format LaTeX.

Dans un document, ces clés sont utilisées comme renvoi à une référence bibliographique. La chaîne de caractères entre crochets est celle que LaTeX substitue à la clé lors de la compilation.

L'ensemble des références constituant la bibliographie d'un document LaTeX doit être placé dans l'un des fichiers utilisés lors de la compilation. La méthode proposée ici sera utile pour construire ce fichier. Elle permettra de retrouver les références dont les clés associées figurent dans le document, avant de les placer en ordre alphabétique dans le fichier bibliographique.

Pour réaliser le tri des enregistrements multilignes, en l'occurrence des références bibliographiques au format LaTeX, nous distinguerons deux méthodes, d'une part, la méthode de jonction des lignes qui composent un enregistrement, d'autre part, la méthode qui consiste à réaliser un tri indirect par adressage des enregistrements après tri des clés.

La première méthode apparaît comme une méthode *ad hoc* qui permet de satisfaire une contrainte déterminée par un outil disponible : la commande `sort`. La seconde est une méthode générale. Elle consiste à générer un script de commandes à partir d'un ensemble de clés qui auront été triées. La possibilité d'implanter cette méthode est conditionnée par la disponibilité d'un moyen de trouver les enregistrements dans un fichier, de les stocker dans un tampon et de les imprimer dans l'ordre des clés. Nous avons envisagé deux implantations de cette méthode. La première avec l'interprète `ed` dans laquelle les enregistrements sont placés dans un tampon, la seconde avec `awk` dans laquelle les enregistrements sont placés dans un *tableau associatif* (voir le paragraphe 3.3).

Les raisons suivantes nous ont fait opter pour une implantation en **awk** :

- la syntaxe de **awk** est empruntée au langage C; un script **awk** est donc, *a priori*, beaucoup plus facilement lisible qu'un script **ed** qui est par ailleurs réputé pour sa syntaxe cryptique;
- la taille des fichiers contenant les enregistrements que l'on peut traiter avec **awk** n'est pas limitée contrairement à **ed**;
- **awk** est un véritable langage de programmation; il est plus puissant que **ed** et fournit des possibilités de programmation qui se révèlent utiles lorsque l'on cherche à réaliser des optimisations et un traitement des erreurs.

L'implantation proposée ne peut être aussi rapide qu'un programme écrit en C ou dans un autre langage. Mais l'utilisation du présent utilitaire se justifie pleinement si l'on considère que ce que l'on recherche en priorité dans la programmation shell n'est pas tant de disposer d'un programme performant que d'obtenir rapidement une solution que l'on pourra aisément modifier et qui pourra servir éventuellement de prototype à une solution écrite dans un langage plus performant.

Les interprètes de commandes utilisés sont présentés dans la partie qui suit.

3 Le shell et les interprètes sed et awk

La présentation des outils sera limitée aux possibilités exploitées pour le groupement. Les exemples proposés seront directement inspirés par les différentes versions du script qui sont présentées dans la dernière partie du rapport.

3.1 ksh

Le shell [Kernigh 84] est l'interprète de commandes du système d'exploitation **UNIX**. **ksh** (pour **Korn shell**) est la version du shell de **UNIX** que nous utiliserons [1]. Le shell permet d'utiliser les commandes de manière interactive ou dans les groupements contenus dans des fichiers que sont les scripts. C'est cette dernière possibilité que nous exploiterons.

Par rapport aux divers commandes et interprètes de commandes **UNIX**, **ksh** est un méta-interprète : il traite comme des données les commandes appartenant à d'autres interprètes, comme **sed** ou **awk**. En fait, un script **ksh** utilisant d'autres commandes que celles internes à cet interprète subit deux interprétations successives : celle de la part de **ksh** et celle de la part des autres commandes.

ksh fournit un ensemble de structures de contrôle pour grouper les commandes. Outre la séquence le script utilise une autre structure de contrôle : le *tube* (ou **pipe**, dénoté par **|**) qui permet de raccorder deux commandes au moyen de leur entrée/sortie, la première produisant des données consommées par la seconde. Ainsi dans

```
sort ... | sed ...
```

[1] Le script pourra être très facilement adapté à **sh**.

la sortie de la commande de tri **sort** est-elle raccordée à l'entrée de l'interprète **sed**. Ce qui transite entre les deux commandes est appelé un *flot* de caractères qui peut, par exemple, être constitué par l'ensemble des lignes d'un fichier de texte.

Autre structure de contrôle le mécanisme dit *de substitution de commande* permettra, par exemple, de placer directement dans une variable le résultat de l'exécution d'une commande. Ainsi,

```
NBERG='wc -l < /tmp/toto'
```

place le nombre de lignes (`wc -l`, pour **Word Count** et **Line**) du fichier `/tmp/toto` dans la variable **NBERG**.

Comme nous l'avons dit, un script **ksh** peut contenir des scripts du nom des autres interprètes utilisés, comme **sed** ou **awk**. C'est à ces deux interprètes que nous allons maintenant nous intéresser.

3.2 sed

sed [McMahon 79] fait partie des nombreux interprètes de commande de UNIX. Il est utilisé pour faire de l'édition non interactive sur flot de caractères (**sed** est l'acronyme de **Stream EDitor**). Il s'agit d'un *filtre* — tout comme de nombreuses autres commandes **UNIX** utilisées ici — c'est-à-dire une commande avec une entrée et une sortie par lesquelles transite le flot.

sed fonctionne selon un cycle simple : une ligne du flot est placée dans le tampon, les commandes du script **sed** sont appliquées en séquence sur son contenu et la ligne est envoyée sur la sortie.

Un script **sed** est formé d'une séquence de règles du type *patron-action*. Les patrons désignent les adresses des lignes du flot sur lesquelles les actions doivent agir. Ces patrons sont de diverses formes, ce peut être un numéro de ligne ou une *expression régulière*. Une expression régulière constitue un modèle d'un ensemble de chaînes de caractères. Elle définit en compréhension cet ensemble en codant les régularités de structure des éléments qui le composent. Donnons quelques exemples d'expressions régulières. `c*` représente une chaîne de caractères de longueur arbitraire uniquement formée du caractère `c`. `[0-9A-Z]` représente un caractère : un chiffre — compris entre 0 et 9 — *ou* une lettre majuscule prise parmi l'ensemble des lettres de l'alphabet. Le caractère `.` quant à lui désigne n'importe quel caractère. Le caractère `^` placé comme premier caractère à l'intérieur des crochets indique que le ou les caractères présents entre les crochets ne doivent pas apparaître dans la chaîne. Ainsi, `[^c][^c]*` permettra de désigner une chaîne de caractères, constituée d'au moins un caractère et ne contenant pas de caractère `c`.

Un script **sed** pourra donc avoir la forme suivante :

```
/expression régulière 1/ action 1
/expression régulière 2/ action 2
```

les obliques étant utilisés pour délimiter les expressions régulières. La partie action se résume le plus souvent à une seule commande désignée par une lettre unique.

Le fonctionnement d'un script sed peut se décrire à l'aide du pseudo-code suivant :

```
Pour chaque ligne i du flot de 1 à n faire
début
        Pour chaque règle j de 1 à n faire
        début
                si appariement avec patron de la règle j
                alors
                        appliquer action associée sur ligne i
                is
        fin
fin
```

Nous utiliserons deux commandes qui permettent d'insérer une chaîne de caractère dans le flot de caractères de sortie. Il s'agit des commandes i\ et a\. L'illustration ci-dessous en décrit l'usage (le caractère $ désigne la dernière ligne du flot de données).

```
1 i\
chaîne de caractères à placer sur la sortie avant la ligne n° 1
        .
        .
        .
$ a\
chaîne de caractères à placer sur la sortie après la dernière ligne
```

Nous utiliserons également la commande de substitution qui prend le contenu du tampon et le transforme selon des modalités qui peuvent être très élaborées. La forme de cette commande est la suivante :

```
s ?  expression régulière ?  substitut ?  option(s)
```

Le point d'interrogation peut être remplacé par tout caractère au choix. Le substitut est la chaîne de caractères par laquelle on veut voir remplacer la partie du tampon qui s'est appariée à l'expression régulière.

Il est possible de reporter dans le substitut tout ou partie de la chaîne de caractères appariée. Dans le substitut le caractère & désigne la totalité de la chaîne de caractères appariée, alors que \n, avec n compris entre 1 et 9, permet de récupérer une partie de la chaîne appariée avec la $n^{ième}$ sous-expression régulière qui aura été placée entre \(\); lorsque plusieurs sous-expressions (imbriquées ou non) existent, pour déterminer n on compte les occurrences de \(de la gauche vers la droite.

Nous utiliserons l'option g de la commande de substitution. Elle est destinée à itérer la substitution autant de fois que possible sur le contenu du tampon. Ce qui revient à itérer l'appariement autant de fois que possible, sachant qu'il ne peut y avoir de chevauchement entre les appariements.

3.3 awk

Tout comme **sed**, l'interprète **awk** [Aho 88] [2] est un éditeur sur flot de caractères, mais il permet de réaliser une beaucoup plus grande diversité d'opérations que le premier. Il s'agit en fait d'un véritable langage de programmation. Un script **awk** est également composé de règles mais par rapport à **sed** les patrons peuvent être plus élaborés et la partie action est définie selon une syntaxe semblable à celle du langage C. Nous n'aurons à utiliser qu'une part infime des possibilités de **awk**.

awk fonctionne selon un cycle similaire à celui de **sed** mais chaque ligne ou enregistrement lu est placé dans des variables. Par défaut, un enregistrement est constitué d'une ligne unique découpée en une série de champs qui sont placés dans des variables : **$0**, **$1**... **$n**, la totalité d'un enregistrement étant disponible dans le paramètre **$0**.

En redéfinissant le séparateur d'enregistrements (**RS** pour **Record Separator**) à "" les enregistrements deviennent alors les groupes de lignes séparés par des lignes vides.

Nous utiliserons deux types de patrons : les patrons prédéfinis **BEGIN** et **END** et des expressions régulières. Les actions associées aux patrons **BEGIN** et **END** s'exécutent respectivement au début et à la fin du fichier [3]. Les scripts **awk** auront la forme suivante :

```
BEGIN{ prétraitement :  action(s) à effectuer avant le chargement du
       premier fichier }
/ER 1/ { action(s) associée(s) à l'expression régulière 1 }
/ER 2/ { action(s) associée(s) à l'expression régulière 2 }
       .
       .
       .
END{ post-traitement :  actions à effectuer sur la fin du dernier
       fichier ou sur exécution de la commande exit }
```

Le prétraitement consiste souvent en des initialisations. Nous utiliserons le post-traitement pour effectuer l'impression d'un tableau qui aura été rempli durant le traitement défini par les actions associées aux expressions régulières.

Pour l'impression nous emploierons la commande **printf** qui est toujours utilisée avec des indications de présentation des données à imprimer. Par exemple, `printf "%s\n\n"` imprimera une chaîne de caractères (`%s`) séparée par une ligne vide (`\n\n` ou double caractère nouvelle-ligne).

Deux commandes seront utilisées pour des optimisations. La commande **next** permet de commencer un nouveau cycle, à savoir de lire un nouvel enregistrement et d'exécuter pour ce nouvel enregistrement les traitements spécifiés. Avec la commande **exit**, on exécute les actions associées au patron **END** et on termine l'exécution.

awk permet de construire des tableaux associatifs et fournit un ensemble de facilités permettant une manipulation aisée de cette structure de données. Un tableau de ce type pourra, par exemple, être composé des éléments suivants : `ERG[1]`, `ERG[2]`, ... , `ERG[`i`]`, `ERG[`n`]`, avec **ERG** le nom du tableau et i l'indice permettant d'accéder à chaque élément.

[2] Le script présenté n'utilise pas les possibilités de la nouvelle version de **awk** présentée dans cette référence. On pourra consulter [Kernigh 85] pour le détail des différences avec la première version.

[3] Si plusieurs fichiers sont donnés en argument à **awk** alors c'est au début du premier fichier et à la fin du dernier que les actions sont exécutées.

On peut utiliser un tableau associatif pour stocker tout ou partie des enregistrements lus. Si dans l'élément i on souhaite placer la totalité d'un enregistrement on écriva une règle dans laquelle figurera l'instruction `ERG[`i`] = $0`.

Pour l'impression de l'ensemble des éléments une boucle avec un compteur utilisé comme indice du tableau suffit :

```
for (i = 1 ; i <= n ; i++) printf "%s\n\n", ERG[i]
```

L'écriture `i++` effectue l'incrémentation du compteur. Elle a la même fonction que `i = i + 1` mais elle est plus concise et plus rapide.

4 Application

Après une présentation du script centrée sur la description de son flux de données nous donnerons trois versions du script. Nous présenterons tout d'abord une version qui dispose du minimum pour être opérationnelle. Ensuite nous verrons comment cette version peut être optimisée en particulier en exploitant le fait que le nombre de clés peut être très inférieur aux nombres d'enregistrements. Enfin nous parlerons de la mise en œuvre effective du script proposé en signalant divers aménagements qui pourront être réalisés pour le confort d'utilisation et l'adaptation aux situations spécifiques qui pourraient devoir être traitées.

Dans ce qui suit nous supposerons que l'on dispose déjà des clés. Nous ne nous intéresserons pas au problème de leur extraction d'un document ou d'une base de données bibliographiques.

4.1 Description

Le principe sur lequel repose le groupement de commandes est celui de la génération automatique d'un script à partir de données, en l'occurrence de clés. Au cours de l'exécution, il y a un changement de statut d'une information manipulée, à savoir les clés : ce sont d'abord des données puis les éléments d'un script.

Le flux de données de la première version du script est représenté à la figure 2. Nous ignorerons les opérations en dehors du cadre en trait plein qui matérialise les limites du script décrit ici. Les données contenues dans un fichier apparaissent dans un cadre en lignes brisées, les opérations dans un cadre en trait plein. Les clés pourront provenir d'un document ou d'une base de données dont on voudra trier l'ensemble des enregistrements. L'existence de cette alternative est indiquée par des flèches en pointillés.

Les clés sont tout d'abord mises en ordre alphabétique. Ensuite, une première opération en compte le nombre qui correspond au nombre d'enregistrements qui doivent être trouvés. Une seconde opération associe à chaque clé un nombre ordinal : l'ordre alphabétique est codé par l'ordre des nombres entiers. Ces nombres seront utilisés pour numéroter les cases d'un tableau associatif dont chacune contiendra un enregistrement. L'utilisation d'un nombre ordinal comme indice du tableau est une facilité qui permettra d'utiliser une boucle avec une variable numérique pour imprimer le contenu de ce tableau.

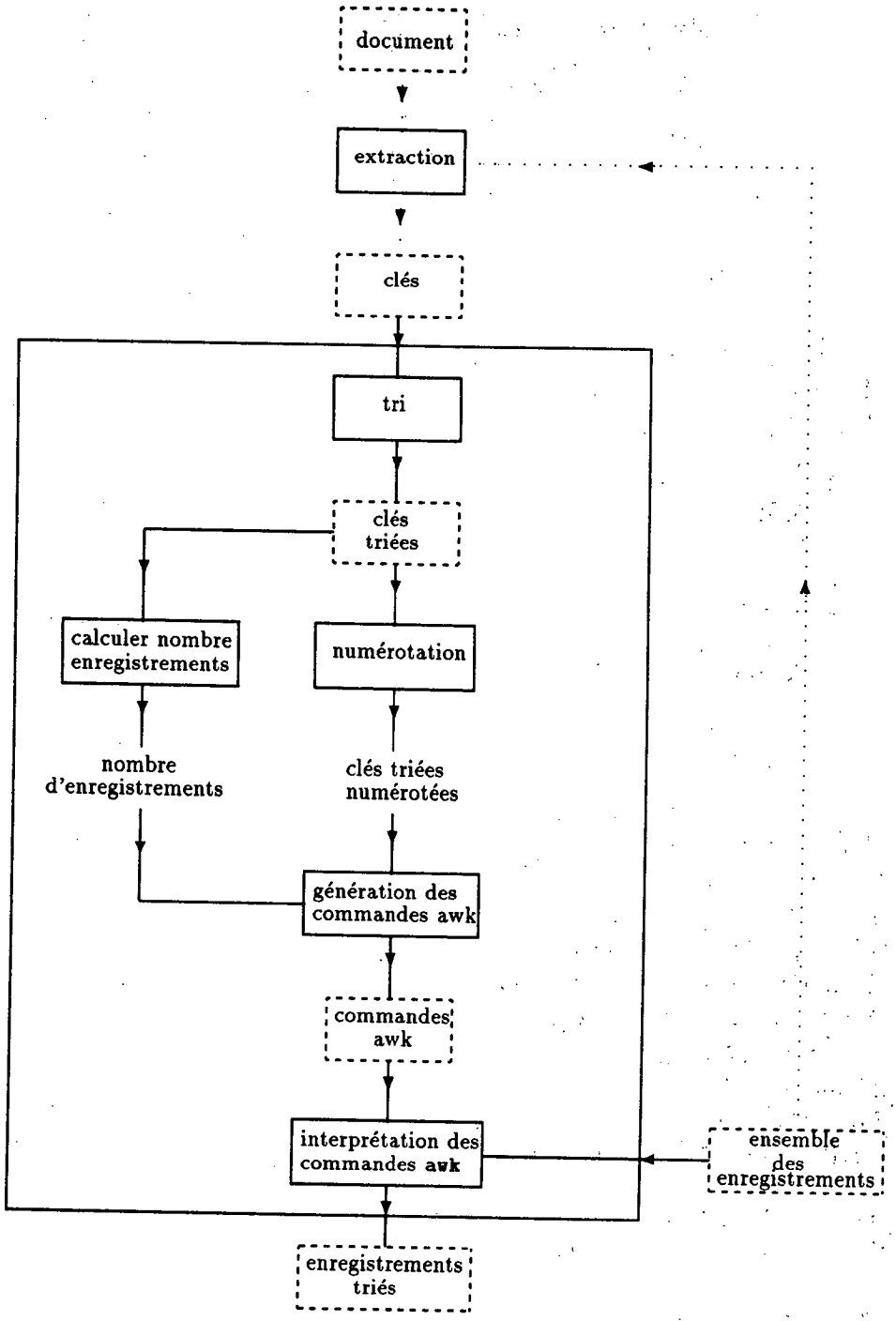

Figure 2: flux de données du script.

```
BEGIN{ RS = "" ; NBERG = 3 }
 /tata/       { ERG[1] = $0 }
 /titi/       { ERG[2] = $0 }
 /toto/       { ERG[3] = $0 }
END{ for(i = 1 ; i <= NBERG ; i++) printf "%s\n\n", ERG[i] }
```

Figure 3: un exemple de script **awk** généré à partir de trois clés.

Le script **awk** est généré à partir de deux types de données, d'une part, le nombre d'enregistrements, d'autre part les clés. Un script généré à partir d'une liste de trois clés exemples est présenté à la figure 3. Il se compose de trois parties.

1. Une partie *initialisation* : l'action associée au patron **BEGIN** permet, d'une part, de redéfinir le séparateur d'enregistrement en indiquant qu'il s'agit d'une ligne vide, d'autre part, d'indiquer quel est le nombre d'enregistrements (**NBERG**).

2. Une partie *recherche et affectation* : on reconnaît l'enregistrement associé à une clé et on le place dans la case du tableau qui lui correspond. Il y a autant de lignes de commande de ce type qu'il y a de clés.

3. Une partie *impression* : l'action associée au patron **END** imprime le contenu du tableau.

Dans le script **awk** le lien entre une clé et l'enregistrement qui lui correspond est explicité sous deux formes successives, d'une part, sous la forme d'un lien de *modèle à référent*, d'autre part, sous la forme d'un lien de *contenant à contenu*. Le premier consiste à écrire une expression régulière, modèle d'un champ de l'enregistrement que l'on manipule, le second consiste à créer un tableau associatif dont le numéro de chacun des éléments est un nombre ordinal qui permet de ranger dans le bon ordre chacun des enregistrements en attendant qu'il soit imprimé.

La dernière opération réalisée par le script est celle que l'on vise, à savoir le tri alphabétique de tout ou partie d'un ensemble d'enregistrements. Pour réaliser l'opération l'interprète **awk** est invoquée avec deux arguments, d'une part, le nom du fichier de commandes **awk**, d'autre part, un ou plusieurs noms de fichiers contenant les enregistrements.

À l'issue de l'exécution du script **awk** les enregistrements triés sont envoyés sur la sortie du script. Il sera tout à fait possible d'effectuer des traitements à la suite ou de récupérer directement les références dans un fichier.

4.2 Version de base

La figure 4 présente une première version du script.

Pour imprimer l'ensemble des éléments du tableau, on doit déterminer le nombre d'éléments qu'il contient. Autrement dit connaître le nombre de clés. Ligne 7 du script, ce nombre est déterminé par la commande `wc -l` dont on place le résultat dans une variable shell (`NBREF`) à l'aide du mécanisme de substitution de commande. Cette variable a été déclarée comme étant du type entier par la commande `typeset`. Pour connaître le nombre de lignes on n'a pas utilisé l'écriture `wc -l /tmp/cles.$$` qui au nombre de lignes ajouterait le nom du fichier. On a eu recours à l'astuce qui consiste à ne plus donner le nom du fichier en argument mais à envoyer le contenu du fichier sur l'entrée de `wc`, ce que dénote le symbole `<` [4].

Les clés sont triées par `sort` :

— en ne retenant pour le tri que les lettres de l'alphabet, les chiffres et les caractères de tabulation (option `-d`);

— en confondant minuscules et majuscules (option `-f`);

— en retenant un seul exemplaire d'une clé (option `-u`);

— et en utilisant le même fichier pour l'entrée et la sortie (option `-o`).

```
 1  trap 'rm -f /tmp/*.$$' 0 1 2 15
 2
 3  typeset -r BIB=/u/mirsa/2/gould/side/verdret/BIB
 4  typeset -i NBERG=0
 5
 6  sort -d -f -u /tmp/cles.$$ -o /tmp/cles.$$
 7  NBERG=`wc -l < /tmp/cles.$$`
 8
 9  awk '{ print $0" "NR }' /tmp/cles.$$ |
10  sed '
11  1i\
12  BEGIN{ RS = "" ; NBERG = '$NBERG' }
13  s#\(.*\) \(.*\)#/{kn:\1}/ { ERG[\2] = $0 }#
14  $a\
15  END{ for(i = 1 ; i <= NBERG ; i++ ) printf "%s\\n\\n", ERG[i] }
16  ' > /tmp/awk_cmd_1.$$
17
18          awk -f /tmp/awk_cmd_1.$$ $BIB/*
```

Figure 4: la première version.

La ligne 9 est un petit script `awk` qui permet d'associer à chacune des clés le numéro de la ligne (contenu dans le paramètre prédéfini `NR`, pour `record number`) sur laquelle elle se trouve. Un caractère blanc est utilisé pour séparer une clé de son numéro ordinal associé.

[4] Il sera également possible d'utiliser l'écriture plus lourde et moins performante : `cat /tmp/cles.$$ | wc -l`.

Le script **sed** se compose de trois lignes responsables de la génération des trois parties du script **awk**. La ligne 15 permet de créer la partie du script **awk** responsable du remplissage du tableau associatif. Ligne 13 le caractère blanc est utilisé comme caractère permettant de distinguer les deux champs s'appariant respectivement aux deux sous-expressions régulières (.*).

Dans l'écriture des expressions régulières associées à chacune des commandes de **awk** qui réalisent les affectations, on aura intérêt à spécifier le contexte dans lequel figure une clé de manière à ce que la désignation de l'enregistrement associé à cette clé soit non ambiguë . Pour les références bibliographiques au format LaTeX, nous utilisons {kn:*clé*}. Pour **awk** il n'existe pas de contraintes sur l'emplacement de la clé dans un enregistrement, si bien que n'importe quelle sous-chaîne d'un enregistrement pourra convenir.

Les commandes générées sont placées dans un fichier temporaire. Pour dénommer ce fichier on utilise une technique qui consiste à suffixer son nom par le numéro du processus shell qui exécute le script. Cette technique permet d'éviter toute confusion avec d'autres fichiers temporaires qui pourraient avoir été créés par d'autres scripts. La commande **trap** sur reception des signaux indiqués permet d'exécuter la commande **rm** qui procède à un nettoyage en effaçant les fichiers temporaires qui ont été créés. Les signaux pris en compte sont la fin normale de l'exécution (indiquée par 0), la déconnexion (indiquée par 1), l'interruption demandée par pression de la touche **delete** (indiquée par 2), ou l'interruption demandée par la commande **kill** (qui envoie par défaut le signal 15). L'option **-f** de la commande **rm** permet de ne pas avoir de message d'erreur en cas de problèmes. Par exemple, si le fichier que l'on cherche à effacer n'existe pas.

À l'issue de la génération de ces commandes, l'interprète **awk** est appelé avec deux arguments, d'une part, le fichier de commandes, d'autre part, l'ensemble des fichiers constituant une base de données (pour le shell le caractère * désigne l'ensemble des fichiers contenu dans un répertoire [5]). Cette base de données se trouve en l'occurrence, dans un répertoire dont le nom est placé dans une variable shell (déclarée comme une variable que l'on ne peut pas modifier grâce à l'option **-r** de la commande **typeset**).

4.3 Optimisations

La seconde version comporte quatre optimisations. La plupart s'effectuent au moyen d'aménagements très simples.

1. La première optimisation sera utile dans le cas où il s'agit de construire une bibliographie. Lorsque l'on a trouvé un nombre de références égal au nombre de clé on interrompt la recherche des références dans la base de données. Une condition importante pour la mise en œuvre de cette optimisation sera qu'il n'y ait qu'un seul et unique enregistrement par clé.

2. Lorsque l'on a trouvé l'enregistrement correspondant à une clé, on stoppe le balayage des règles. Autrement dit on commence un nouveau cycle.

[5] À l'exception des fichiers dont le nom commence par le caractère point.

3. On n'attend plus d'avoir trouvé tous les enregistrements avant de commencer l'envoi des enregistrements sur la sortie : on les envoie sur la sortie dès que l'on dispose dans le tableau d'une suite continue d'enregistrements et que ceux qui précèdent la suite ont déjà été imprimés.

4. Lorsque l'on a imprimé un enregistrement, il est immédiatement effacé. Une telle optimisation, en libérant la mémoire, pourra être utile si l'on doit traiter une base de données importante.

Le script **awk** optimisé apparaît à la figure 5.

```
BEGIN{ RS = "" ; NBERG = 3 ; CP = 0 ; ORD = 1 }
ERG[ORD] != "" { while (ERG[ORD] != "") {
                 printf "%s\n\n", ERG[ORD]
                 delete ERG[ORD]
                 ORD++
                 }
}
/tata/ { ERG[1] = $0 ; CP++ ; if (CP == NBERG) exit ; next }
/titi/ { ERG[2] = $0 ; CP++ ; if (CP == NBERG) exit ; next }
/toto/ { ERG[3] = $0 ; CP++ ; if (CP == NBERG) exit ; next }
END{ for (i = ORD ; i <= NBERG ; i++) printf "%s\n\n", ERG[i] }
```

Figure 5: le script **awk** optimisé.

Le compteur CP est utilisé pour compter le nombre d'enregistrements trouvé. Lorsque CP est égal au nombre de clés, la commande **exit** est exécutée, l'impression des enregistrements réalisée et l'exécution stoppée. La commande **next** est chargée de faire démarrer un nouveau cycle lorsqu'un enregistrement a été trouvé.

Les deux dernières optimisations se traduisent par l'ajout d'une règle au script. Le compteur ORD permet de conserver la position du dernier éléments du tableau qui a été imprimé. La boucle **tant que** imprime les éléments depuis cette position jusqu'au prochain élément qui n'a pas encore été affecté. La commande **delete** permet d'effacer un enregistrement après son impression. La présence de la boucle itérative associée au patron END est obligatoire pour imprimer les enregistrements qui ne l'ont pas encore été au moment où l'on trouve le dernier enregistrement.

La version optimisée est donnée à la figure 6.

4.4 Adaptation et aménagements

Dans cette dernière partie nous considérons diverses possibilités d'adaptation de la méthode aux situtations spécifiques qui pourraient devoir être traitées, ainsi que des aménagements destinés à améliorer la robustesse du script.

```
 1  trap 'rm -f /tmp/*.$$' 0 1 2 15
 2
 3  typeset -r BIB=/u/mirsa/2/gould/side/verdret/BIB
 4  typeset -i NBERG=0
 5
 6  sort -d -f -u /tmp/cles.$$ -o /tmp/cles.$$
 7  NBERG=`wc -l < /tmp/cles.$$`
 8
 9  awk '{ print $0" "NR }' /tmp/cles.$$ |
10  sed '
11  1i\
12  BEGIN{ RS = "" ; NBERG = '$NBERG' ; ORD = 1 ; CP = 0 } \
13  ERG[ORD] != "" { while (ERG[ORD] != "") { \
14                   printf "%s\\n\\n", ERG[ORD]  \
15                   delete ERG[ORD] \
16                   ORD++ \
17                   }\
18  }
19  s#\(.*\) \(.*\)#/{kn:\1}/{ERG[\2] = $0 ; CP++ ; if (CP == NBERG) exit ; next}#
20  $a\
21  END{ for(i = ORD ; i <= NBERG ; i++) printf "%s\\n\\n", ERG[i] }
22  ' > /tmp/awk_cmd_1.$$
23
24          awk -f /tmp/awk_cmd_1.$$ $BIB/*
```

Figure 6: la version optimisée.

Une variante du script **awk** est présenté à la figure 7. Elle sera utile si l'on désire trier la totalité des références contenues dans une base de données. Cette fois les indices du tableau associatif sont définis à partir de l'un des champs de l'enregistrement. Dans l'exemple proposé on a supposé que le caractère séparateur de champ était | et que la clé de tri était le second champ d'un enregistrement. La première version de **awk** n'accepte qu'un seul caractère comme séparateur de champ. La nouvelle version ([Aho 88]) permet de définir le séparateur de champ à l'aide d'une expression régulière. Pour obtenir une définition du champ qui correspond à la clé que nous avons jusqu'à présent utilisée on pourra réaliser l'affectation suivante **FS="[kn:|]"** (le caractère | dénotant une alternative). Si jamais on désirait n'extraire qu'un sous-ensemble de la base de données il sera toujours possible d'associer une expression régulière à l'unique règle chargée de la création du tableau associatif.

```
BEGIN{ RS = "" ; FS = "|" }  #Si | est le sep. de champs
       { ERG["$2"] = "$0" }  #Si $2 est la cle
END{
 printf "%s\n\n", ERG["tata"]
 printf "%s\n\n", ERG["titi"]
 printf "%s\n\n", ERG["toto"]
}
```

Figure 7: un script **awk** pour trier la totalité d'une bibliographie

Dans cette variante le tri n'est plus obtenu en rangeant à un endroit prédéfini chaque enregistrement mais en réalisant l'impression des enregistrements dans l'ordre prédéfini par l'ensemble des commandes d'impression associé au patron **END**.

Les versions précédentes du groupement ont une limite qu'il peut être intéressant de dépasser. En effet, les clés ne peuvent contenir des caractères, comme \ et /, qui appartiennent au lexique de **awk** et les caractères du langage des expressions régulières, comme ? et *. Si jamais on voulait faire figurer les caractères \ et / dans les clés il suffira d'ajouter comme première ligne du script **sed** :

```
s#[\\/]#\\&#g
```

Cette commande remplacera toute occurrence de l'un des deux caractères considérés par ce caractère, précédé d'un contre-oblique. Il n'est pas nécessaire de tenir compte du contexte dans lequel figure un caractère pour savoir s'il est ou non un métacaractère, et si donc il doit être littéralisé, puisque la littéralisation remplace les deux caractères **contre-oblique [méta]caractère** par **caractère**.

Un certain nombre d'aménagements devront être effectués pour assurer, en toutes circonstances, le bon fonctionnement du script. On devra, par exemple, veiller à ce que

la syntaxe de la base de données soit conforme à celle exigée, en particulier à la présence d'une ligne vide entre chaque référence [6] et à l'unicité des références.

La figure 8 donne une version du script pour laquelle on n'a réalisé deux aménagements permettant de contrôler le bon fonctionnement du script. Un premier aménagement est destiné à ne permettre qu'une seule affectation d'un élément du tableau ce que l'on exprime très facilement en testant si l'élément du tableau est vide ou non (ligne 20). Il faut également ajouter comme condition que le compteur `ORD` soit inférieur au numéro de l'enregistrement car il se pourrait très bien que l'enregistrement ait déjà été trouvé et effacé.

Un second aménagement indique à l'utilisateur s'il y a des clés pour lesquelles on a pas trouvé d'enregistrement. Pour chaque élément du tableau qui n'a pas été rempli le script **awk** génère une commande **sed** de la forme *n*p qui permet d'imprimer la clé qui apparaît à la ligne *n* dans le fichier contenant l'ensemble des clés. À l'issue du tri, s'il manque effectivement des enregistrements, le script **sed** est exécuté(lignes 34 à 38). On remarquera que **sed** est appelé avec l'option `-n` qui supprime l'impression par défaut. L'option `-u2` de la commande `print` et `1>&2` permettent d'envoyer les données sur la sortie erreur.

Tel qu'il est proposé le script n'a pas d'argument. Dans une version personnalisée du script il sera toujours possible de lui passer en argument des informations, par exemple, le nom du fichier qui contient les clés ou le nom du répertoire qui contient la base de données bibliographiques.

Signalons enfin que le groupement proposé possède une limite qui est la taille maximale d'un enregistrement que **awk** peut lire ou imprimer avec la commande `printf`. Cette limite est de 3000 caractères [Aho 88].

5 Conclusion

La méthode sur laquelle repose le groupement décrit pourra très simplement être adaptée pour réaliser d'autres opérations que celle présentée. On pourra l'utiliser, par exemple, pour la fusion ou la jointure de deux ensembles de données.

Nous l'avons utilisée pour construire la bibliographie de ce rapport dans laquelle chaque référence est suivie des numéros de page du texte où elle est citée. Cette bibliographie est fabriquée à partir de la fusion de deux types de données associés à une clé, d'une part, une référence bibliographique, d'autre part, les numéros de page où se trouve citée cette référence.

Si les adaptations sont dans leur principe souvent faciles à réaliser il est conseillé de procéder méthodiquement. Le programmeur aura ainsi interêt à travailler sur le seul script **awk** avant de penser à l'intégrer dans le générateur de commandes, à savoir le script **sed**.

Disposer d'outils est une chose, mais savoir de quelle manière on va pouvoir les associer pour obtenir l'effet souhaiter est une tout autre affaire. Le groupement proposé par sa simplicité et sa concision devrait pouvoir utilement se joindre à l'ensemble des groupements et méthodes dont dispose déjà le programmeur shell.

[6] La présence d'une ligne vide en fin de fichier n'est pas obligatoire.

```
 1  trap 'rm -f /tmp/*.$$' 0 1 2 15
 2
 3  typeset -r BIB=/u/mirsa/2/gould/side/verdret/BIB
 4  typeset -i NBERG=0
 5
 6  sort -d -f -u /tmp/cles.$$ -o /tmp/cles.$$
 7  NBERG=`wc -l < /tmp/cles.$$`
 8
 9  awk '{ print $0" "NR }' /tmp/cles.$$ |
10  sed '
11  1i\
12  BEGIN{ RS = "" ; NBERG = '$NBERG' ; CP = 0 ; ORD = 1 } \
13  ERG[ORD] != "" { while (ERG[ORD] != "") { \
14                   printf "%s\\n\\n", ERG[ORD] \
15                   delete ERG[ORD]        \
16                   ORD++ \
17                   } \
18  }
19  s#\(.*\) \(.*\)#\
20  /{kn:\1}/{ if (ERG[\2] == "" \&\& ORD <= \2) { \
21                   ERG[\2] = $0 ; CP++ ; if (CP == NBERG) exit ; next \
22                   } \
23          }#
24  $a\
25  END{ for (i = ORD ; i <= NBERG ; i++) { \
26          if (ERG[i] == "") print i"p" > "/tmp/manquants.'$$'" \
27          else printf "%s\\n\\n", ERG[i] \
28          } \
29  }
30  ' > /tmp/awk_cmd_1.$$
31
32          awk -f /tmp/awk_cmd_1.$$ $BIB/*
33
34  if test -s "/tmp/manquants.$$"
35  then
36          print -u2 "$0 : cles sans enregistrement : "
37          sed -n -f  /tmp/manquants.$$ /tmp/cles.$$ 1>&2
38  fi
```

Figure 8: une solution complète.

Remerciements

Ce travail a bénéficié de nombreux échanges avec Francis Montagnac, Alain Giboin et Henry Borron, que je remercie très vivement.

Bibliographie

Chaque référence est suivie des numéros de pages où elle est citée.

[Aho 88] AHO, A.V., KERNIGHAN, B.W. & WEINBERGER, P.J. : *The* **awk** *Programming Languages.* Addison-Wesley Publishing Compagny, 1988.
pages : 4, 9, 17, 18

[Aho 86] AHO, A. & SETHI R. : Maintaining Cross References in Manuscripts. *Software-Practise and Experience*, **18**(1), 1-13, Jan. 1986.
pages : 4

[Bentley 85] BENTLEY, J. : Programming Pearls. *Communications of the ACM*, **28**(6), Jun. 1985.
pages : 4

[Kernigh 85] KERNIGHAN, B.W. : **awk** as a General-Purpose Programming Language. *Proceedings of the EUUG Conference*, Copenhagen, Denmark Bella Conference Centre, 10-12 September 1985.
pages : 9

[Kernigh 84] KERNIGHAN, B.W. & PIKE, R. : *The* **UNIX** *programming environment.* Englewood Cliffs, New Jersey : Prentice-Hall, 1984.
pages : 6

[Knuth 73] KNUTH, D.E. : *Sorting and searching : The Art of Computer Programming*, **3**. Addison-Wesley, Reading MA, 1973.
pages : 4

[McMahon 79] MCMAHON, L.E. : **sed** — *a noninteractive text editor.* Computing Science Technical Report 77, AT & T Bell Laboratories, Murray Hill, N.J., 1979.
pages : 7

Imprimé en France
par
l'Institut National de Recherche en Informatique et en Automatique